Fresh Object System Refactor Notes

# Object Id

Create new type ObjectId that specifically identifies an object via a class name plus object name. This supplements the similar idea in AssetManager (AssetId).

# Object Id Uniqueness

The uniqueness system has been through some variations. Originally names (or ids?) were required to be unique. In honor of Flash I removed this rule. But then I needed some degree of uniqueness so I added the CreateOrGetUniqueObject() system. Rationale: (1) Without such a system, when you Create an object you will always create it. To avoid this you could have CreateOrGetObject(), but this still implies that object ids are at least sometimes unique. So, for example, there’s ambiguity about whether Object\* GetObject( className, objectName ) makes sense. (2) Implementing GetObject() as above with a list is relatively slow. Not necessarily a problem, but an additional mapping system seemed prudent.

I’ve now got a hybrid system, with both non-unique and unique objects. But this is really a hodgepodge. Moreover, it’s complicated by the additional layer of another object system: the “Assets”.

Finally, for the new serialization system, objects really *should* be uniquely ID’ed. Therefore I will now refactor to require unique IDs (not names!) for every object. This is a break with Flash but who cares? It’s in line with Unreal, at least, if I’m looking for precedents.

# Object Construction

The construction system has been a pain from the start. The underlying system requires—for good reason—that every object support a simple constructor that takes only a name. This allows objects to support the CreateObject() factory-based construct-anything system. Unfortunately objects often want other parameters on their constructors. This causes the need for either two fully-formed constructors (highly error-prone) or a bogus “conformance” constructor. Ouch!

The solution I want to move to is one that allows you to pass an “ObjectInitializer” object to any object constructor as well as to an Initialize() function. ObjectInitializer amounts to a map of property names to property values. The Object base class automatically handles using the initializer to initialize all registered properties. An initializer can legally hold as many property values as it likes: it can have missing properties as well as extra ones.

# Default Properties

The issue of how to deal with default values has proven consistently befuddling and painful. There are basically two options:

* Default values are applied to class members *before* the constructor is called. Thus the constructor can override defaults. Loading (e.g. from Xml or an ObjectStore) happens after the constructor and so has the most power. This is the **Unreal** solution.
* Default values are applied *after* the constructor is called but before loading. Thus, default values override the constructor.

The latter system has basically proven unworkable. It causes objects to be initialized very differently depending on whether they’re created with new or ObjectManager::CreateObject(). It’s also very unintuitive to write a bunch of code into a C++ constructor only to have your class end up mysteriously have different values a moment later. Constructor code feels like it be authoritative. Finally, constructor code wants to depend on default values, not be depended on by them. Default values is a lower-level concept.

Therefore the former system works better on a semantic level. The implementation is scary though: I load default values by using template magic, then call the constructor afterward using placement new. This closely maps to Unreal, which makes it feel well-established and relatively safe.

But there is at least one big problem with this approach as well. Although it works great for ints, floats, and other built-in types (as well as simple user-defined types like vec3s), it doesn’t work well for std::string and the like. You can’t simply assign a default value to a std::string variable and have it work. The string has to be constructed.

The workaround is to use placement new *again*, now for each property in the class, prior to the constructor being called. But now this defeats the point of default properties, because when the *real* object constructor is called, the std::string (or whatever) object’s constructor is called again, now without a default value (and probably squashing the former object without bothering to destruct it first).

It’s difficult to imagine how to repair the system unless you could somehow automatically detect and compile-time reject attempts to give “complex” types a default value.

It seems to me that the best solution is just to withdraw from default values entirely. Just get rid of them completely. We could still automatically zero-out values if that seems helpful. But no other default value is provided.

I’m not sure whether this would actually violate anything. I don’t really know why I introduced default values in the first place. It might have just been in imitation of Unreal, or it might have served a real, clear need.

In any case, the fact that I support arbitrary data member types (unlike Unreal) means that providing default value support for those data is unrealistic (ha ha).

TODO connection to XML saving/loading system